

The Multi-Core Era - Trends and Challenges

Peter Tröger, Blekinge Institute of Technology

October 30, 2008

Abstract

Since the very beginning of hardware development, computer processors were invented with ever-increasing clock frequencies and sophisticated in-build optimization strategies. Due to physical limitations, this 'free lunch' of speedup has come to an end.

The following article gives a summary and bibliography for recent trends and challenges in CMP architectures. It discusses how 40 years of parallel computing research need to be considered in the upcoming multi-core era. We argue that future research must be driven from two sides - a better expression of hardware structures, and a domain-specific understanding of software parallelism.

1 Introduction

In 1995, Gregory F. Pfister formulated three ways of doing anything faster [38]:

- Work harder.
- Work smarter.
- Get help.

The original statement explained the benefits of parallel processing on cluster infrastructures. Instead of relying on the bounded scalability of a single machine ('work harder'), he proposed the usage of multiple machines working together as a cluster ('get help'). Within the last years, it turned out that Pfister's principles now also apply to a different area.

Since the very beginning of hardware development, computer processors were invented with ever-increasing clock frequencies ('work harder') and sophisticated in-build optimization strategies ('work smarter'). Developers and industry got used to the fact that applications were simply getting faster by a simple exchange of hardware. Moore's law about the ever-increasing number of transistors per chip still applies, but brings no longer a better computational performance per default. The traditional way of increasing clock rates and higher packaging densities reached its physical limitation in cooling and power management [37].

In order to keep the promise of faster processing by hardware exchange, all CPU vendors now use the additional transistors for *chip multi-processing (CMP)* architectures, which are currently named as *multi-core* or *many-core* processor design. A multi-core CPU combines multiple independent *execution units* into one processor chip, in order to execute multiple instructions in a truly parallel fashion ('get help'). The cores of a CMP processor are sometimes also denoted as *processing element* or *computational engine*. According to Flynn's taxonomy [14], the resulting systems are true multiple-instruction-multiple-data (MIMD) machines, able to process multiple threads of execution at the same time.

Achieving processing speedup by using parallel execution units is obviously nothing new. The concept of the first *multiprocessor computer* goes back to the 60's with initial architectures such as the ILLIAC IV [5]. The resulting challenges were also discussed to large extend, sometimes even 25 years ago [27].

What is the difference ?

The true paradigm shift today is not the realization of CMP architectures, but their spreading in all computer markets. Embedded systems, mobile phones, desktop systems and server systems now include multiple cores out of the box. A parallel computer is no longer a dedicated hardware setup for special purposes.

It is commodity.

While parallel computing of the past was only intended for a specific set of problem domains, it is now relevant for every scientific, industrial or private application running on a computer. Parallel computing now becomes visible for millions of industrial software developers in practice, who usually lack experience in the creation and handling of fine-grained parallel activities. For academia, the shift will influence both future teaching and research in computer science and software engineering. Many traditional areas such as computer hardware architecture, programming languages, design patterns, scheduling theory and parallel algorithms will gain more attention in the future, since the upcoming research challenges demand answers from these fields. In the following text, we want to provide a high-level overview about some of the identified issues in this area. We base the argumentation on one fundamental statement:

"40 years of parallel computing need to be considered."

2 Parallel Hardware

The support for multiple parallel activities (*multithreading*) in hardware is realized on different levels in today's computer systems (see Figure 1). The scheduling of parallel activities must now consider this given 'stack' of execution units. Even though modern operating systems are meanwhile aware of this effect [53], they still cannot consider data dependencies between parallel activities in their scheduling decision. The application therefore has to support the operating system scheduler in the placement with its specific knowledge [42].

On the lowest level, the execution unit itself can have a super-scalar architecture. A hardware-controlled parallel usage of execution unit components enables the execution of multiple processor instructions during one clock cycle. This approach of *instruction-level parallelism (ILP)* is limited [13], but widely realized approach in modern processor designs.

Each execution unit can additionally support the concept of a logical processor, which allows a *simultaneous multi-threading (SMT)* inside the processor pipeline [50, 12]. This approach allows to hide data access latencies for some of the threads, by utilizing the execution unit for other tasks during a blocking period. This realizes a virtual sharing of one execution unit within a processor. SMT is better known under the marketing term *hyperthreading* from Intel [32]. It maintains the architectural state of the execution unit (mostly CPU registers) separately per logical processor, in order to allow context switching in hardware.

A set of execution units can be put together to form a *chip multi-processing (CMP)* architecture [35]. Most such processors contain separate L1 caches for each of the units, and a shared L2 cache for the whole processor chip. CMP forms the latest trend in processor design, even though earlier attempts already gained some experience with this approach [51].

The different ways of exploiting parallelism inside one processor chip are then summarized as *chip multi-threading (CMT)* capabilities [42]. The next higher level is *symmetric multi-processing*

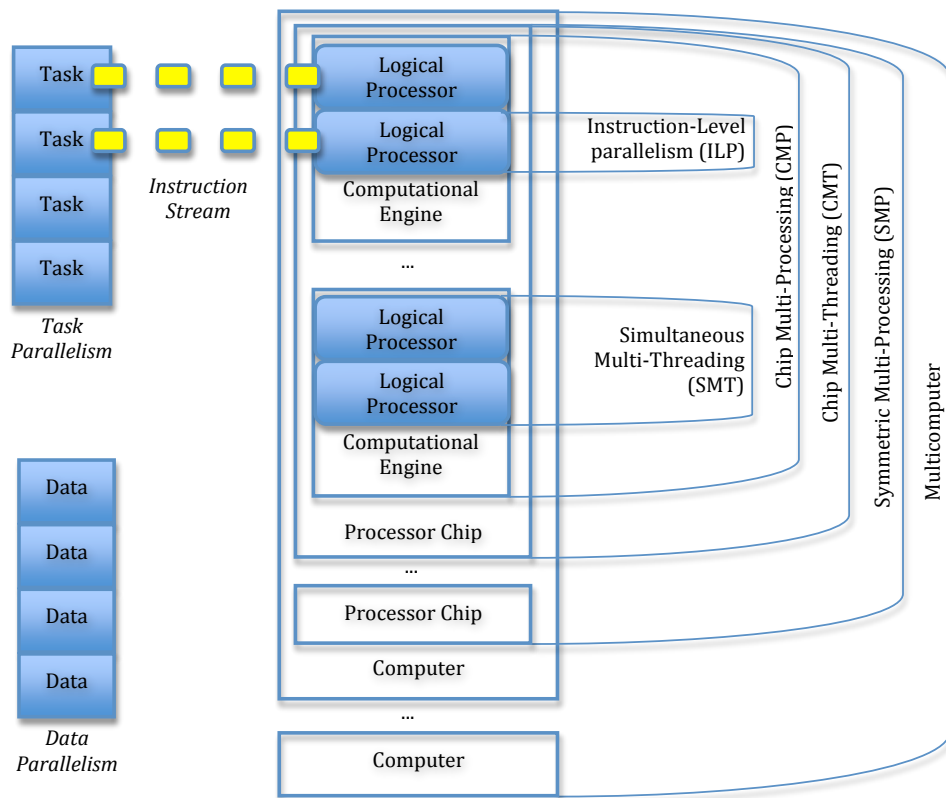


Figure 1: Hardware parallelism hierarchy

(SMP) with multiple processors, and ultimately the realization of a computer cluster as one *multicomputer*.

The widely promoted new era of multi-core systems basically focuses on the introduction of CMP features in standard desktop processors. Even though there is a high amount of daily news on recent hardware development, some common trends can be identified in this development.

Most sources agree that the number of execution units per chip will be in the magnitude of thousands in the next 5-10 years. A recent report from Berkeley [4] predicts CMP processors with thousands of parallel execution units as the mainstream hardware of the future. The "Intel Terascale Computing" initiative designed in 2008 an 80-core research prototype to investigate future challenges for the company. The MIT-originated TILE64 processor architecture today provides a grid of 64 low-speed (866 MHz) execution units on a chip, interconnected by a mesh network. In general, the trend with such *homogeneous multi-core architectures* occurs to be a high number of less complex execution units, working together by some high-speed connection grid.

The opposite approach is a *heterogeneous multi-core architecture*. One example is the IBM Cell processor. It combines a single master processor element with a set of vector processors. The relevant property here is the heterogeneous set of execution units combined to one chip. It allows an prepared algorithm to use an optimized core for the computational task, while doing different activities in parallel. This kind of task-level parallelism is already common for modern personal computers with their dedicated chips for graphic processing and I/O, but now also reaches the CPU itself. Another well-known example is the zSeries mainframe concept [36], combining different processing units on one *multi chip module* with a shared L2 cache. Some of the units are dedicated as fault tolerance spares or diagnostic chip. This demands specialized compilers and an operating system prepared for the according processor platform.

A still underestimated hardware problem is the connectivity of the execution units. The overall chip still must be connected to memory and I/O devices, which are an order of magnitude slower in their latency time [9, 39]. Industry provides some solutions for this issue, which still need to be evaluated for higher numbers of execution units [48].

The overall idea of many slow processing units connected by a high-speed bus is also not completely new. Computer hardware history shows the INMOS transputer concept [34], classical vector computer computers or massive parallel processing (MPP) installations. It is therefore necessary to consider the tremendous existing knowledge from these areas in the future.

A problem example in the hardware category is the contention of shared resources. With the increased introduction of parallelism in software layers, the processor faces a high number of threads with different memory access patterns and cache utilization profiles. Proposals by Intel suggest the introduction of a QoS-aware memory hierarchy [26], where the operating system prioritizes some threads in their cache and memory bandwidth usage. Other strategies still need to be investigated. This shows the general importance of a *better software to hardware mapping*. Above the level of superscalar instruction processing, all parallelization coordination must now be done in software. This coordination must consider the given parallel hardware layers, without binding itself too much on one particular processor architecture. It is therefore urgent to express modern hardware design in a more generic way, in order to develop appropriate scheduling and data management approaches for parallel applications.

The history of high-performance computing provides the idea of an *abstract MIMD machine model*, which is also applicable for modern CMT architectures. Such models support the run-time performance analysis for a given parallel algorithm, based on a very abstract understanding of a parallel execution environment. Famous examples are the LogP model [11], the bulk synchronous parallel (BSP) computer model [52] and the parallel random access machine (PRAM) model [15].

Most of the existing models imply unlimited space or cycle time, and can even lead to contrary efficiency numbers for the same algorithm [24]. Other models such as the *multicomputer* [18] focus only on distributed systems with multiple processors. It is therefore necessary to find a better abstraction of multi-core hardware architectures. This includes the consideration of timing effects reasoned by caches and mutual exclusion, which more and more turn out to be the performance-relevant factor.

We therefore propose the development of a realistic and feasible *multi-core machine model*, in continuation of the existing approaches. This model can provide the foundation for future research on parallelized algorithms and software design patterns.

3 Parallel Software

Multi-core systems are true parallel computers. The new trend therefore leads to the wider recognition of partially well-known and partially largely software problems. Industry and research already agree upon the fact that the existing *programming paradigms and languages* as well as *design patterns* do not align to modern processor design [21]. This leads to the fact that the original hardware scalability problem is about to be replaced by a software parallelization problem. Challenges that are well known to the parallel computing community are now the problem for every software developer.

This is nothing less than a paradigm shift in education, training, and daily practice of software development. Developers must get a basic understanding of parallel programming from the very beginning, treating a sequential program only as special case. This relates not only to tools and languages, but also to design patterns, algorithmic thinking, testing strategies and software architecture principles. The usage of multiple execution units has even influence on dependability issues [1], meaning that it also becomes a relevant aspect for research on fault-tolerant systems.

Basic Principles

The basic principles of parallel respectively concurrent programming are known for a long time. The mutual exclusion problem, deadlock, livelock and starvation are well-known questions for most computer science students and researchers. The challenge here is the consideration of such problems for the 'average' programmer. Due to the industrialization and wide-spread of software development, more and more developers do not have a solid scientific background in parallel computing. It is therefore necessary to assist these people in their development effort by an according testing mechanism, programming environment and high-level helper functionality.

There are widely accepted principles for the speedup achievable by parallelization. The two corner stones are Amdahls and Gustafsons laws. Amdahls law [2] puts an upper limit on the parallelization speedup achievable with a constant problem size. Figure 2 shows thow speedup achievable by more execution units is always limited, even with a mostly parallelized application (P=95%) . Gustafsons law [22] relaxes this upper bound by expressing that speedup can always be achieved by the shift to bigger problems.

These two intuitive facts are still very valid in the modern multi-core era [25], and therefore need to be considered in all related research. This demands a proper analysis of parallelization strategies for software. We propose that this kind of analysis can be for application domains in a whole, in order to derive generic parallelization principles and approaches. One example for this strategy is the traditional parallel database research, another one is the new area of parallelized XML processing [23, 31]. Future work needs to identify more such application domains and should provide according parallelization strategies for them. This of course demands a heavy

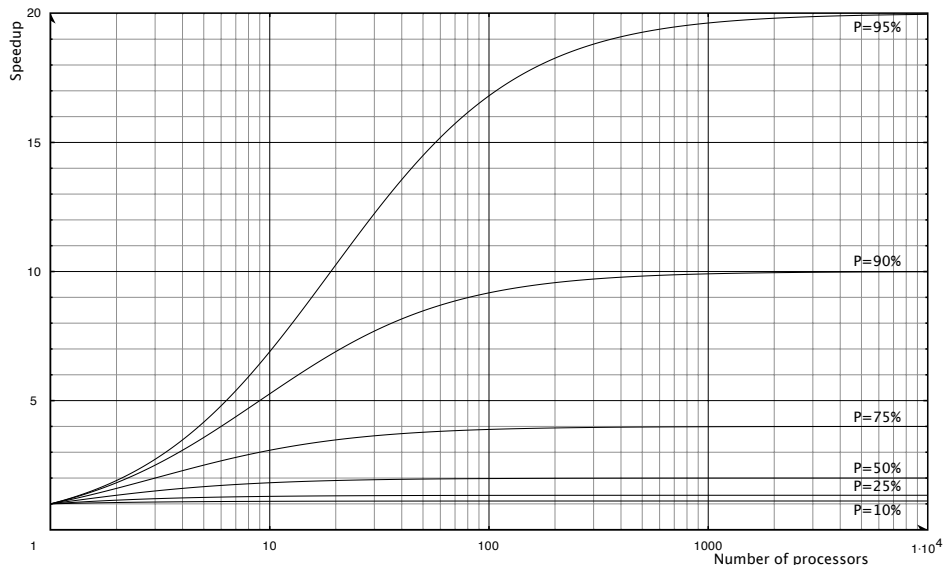


Figure 2: Amdahl's law: Program with parallel portion P

involvement of the according user communities, which is a new chance for computer science to be better connected to other sciences.

Parallelization Support

Parallelization support, already implemented by somebody else, can be provided through the compiler, the operating system or third-party libraries. Such support functionality is used by the application developer either implicitly (e.g. automated loop parallelization) or explicitly through new language constructs respectively a dedicated parallel language.

A compiler can try to formulate the assembler code in a way that multiple execution units are automatically used. In the perfect world, even the operating system developer would then be freed from the consideration of parallel execution strategies. Recent projects in this area [6] still show that this remains a grand challenge of computer science. Since the parallelizing compiler has to determine possible side effects automatically, it becomes extremely hard to still generate valid code. Automatic parallelization mainly focuses on loops, since the identification of coarse-grained parallel chunks remains application-dependent. It is therefore a commonly accepted fact that parallelizing compilers can only provide limited help with speedup by parallelization.

The next level of software parallelism support is the operating system, which currently acts as common glue between hardware parallelism and application parallelism (see Figure 3). All modern systems support preemptive scheduling of parallel activities, commonly named as *threads* [46]. This does not free the application from actually creating and maintaining these parallel activities, but it allows an abstract usage regardless of the particular underlying hardware. Multithreading is still the major parallelization paradigm, support by standardized libraries and virtual runtime environments. While early multi-threading libraries performed the task scheduling by themselves, it is meanwhile common the map application-level parallelism directly to operating system threads.

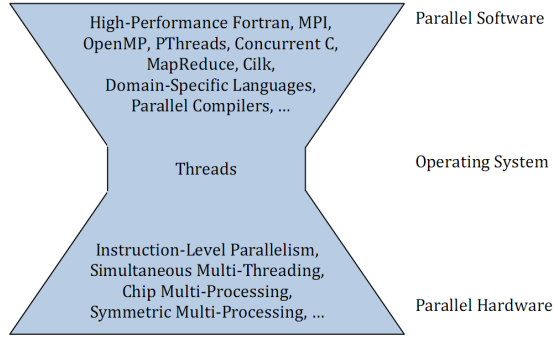


Figure 3: The hourglass of parallelization support

Running more and more software in such a multi-threaded fashion leads to different kind of processing workloads, all with different memory access patterns. One example are virtualization products, where a highly varying load from the guest operating system is mapped to one operating system process in the host system. Iyer et al. [26] predict this to be a major problem for optimal resource utilization. As one possible solution, they propose the prioritization of some threads on hardware level. This allows an optimized usage of cache space and memory bandwidth, but demands according hints from the operating system.

On application level, there is a variety of possibilities to express concurrent activities. The can be roughly categorized in dedicated parallel programming languages and sequential language extensions.

For the large base of sequential programs written in C- or Java-style languages, it is common to use libraries or abstractions for threads. One popular example is the OpenMP [40] language extension, other examples are Cilk [7] and Concurrent C [20].

More radical approaches criticize threads as wrong kind of abstraction, and rely instead on approaches such as pure data-parallel programming [16, 43], functional programming [41] or reactive programming [49]. A third class of researchers elaborates on the notion of *domain-specific languages*, in order to encapsulate the parallelization in high-level language constructs. Popular examples in this category are SQL, MapReduce [29], or Simulink. It remains an open research question which of these approaches is most suitable for the programming of scalable commodity applications based on thousands of execution units. In fact, most implementations still end up in mapping the activities to operating system threads. Alternative solutions mostly demand a virtual runtime environment for the context switching [47].

In general, application developers have to decide upon the degree of control they want to have, starting from low-level locking primitives up to implicit parallelism in functional languages. The low-level approach is still the dominating idea in high-performance parallel computing, since it is the only way to get maximum performance. The more abstract solutions are about to become more popular for the broader mass of developers. The choice influences also the design of according debugging facilities – low-level parallelisation support demands an according support for investigating problematic parallel activities [10].

From a theoretical perspective, the best approach for an parallel application is the usage of a dedicated parallel programming language. Many past initiatives in high-performance computing tried to introduce according solutions, in order to get the maximum benefit from the given compute power. This led to a long history of libraries and parallel programming languages, all trying to abstract basic concurrency and synchronization issues for the developer. A few of them remained successful [16, 17, 28], most of them are forgotten. Future research for multi-core

enabled software needs to remember the successful and - especially - the failed attempts for the abstraction of parallel processing issues, in order to learn from the past. A number of language proposals are still promising [3, 8], and now need to be considered for future programming concepts. At the moment, there is still no agreement about the feasibility of such languages for average industrial developers [45].

Beside the way of expressing parallel activities, it is also still relevant to work on scalable algorithms. Latest research on *lock-free* data structures [19] and software transactional memory [30] shows that there is still a lot of potential in scalable algorithm and data structure design. The high-performance computing field has a long tradition in this area, such as with linear algebra computation, spectral methods, matrix calculations, atmosphere modeling based on partial differential equations, VLSI floorplan optimization or graph traversal [18]. It is therefore necessary to consider this huge body of knowledge for the upcoming era of parallel computers.

4 Summary

Multi-core architectures are meanwhile the state-of-the-art in IT hardware. Computers of the future will contain thousands of execution units per chip, either homogeneous or specialized for particular purposes. The parallelization of application workload is about to become the most relevant strategy for speedup and scaleup in every kind of application. Since single thread performance is no longer given for free, it is unavoidable to think and program in a parallel fashion.

'The free lunch is over.' [44].

The future research for multi-core enabled applications must be driven from two sides - a better expression of hardware, and a better design of software. The increasing variety of CMP processor architectures must be abstracted in better models, in order to allow an appropriate software design. Software on the other side must become analyzable and designable according to the parallel workload it produces. This includes the formulation of design patterns [33] and algorithms prepared for parallel execution. Some applications will be scalable by default - 3D graphics, scientific computing or high-throughput server computing. The interesting challenges are all the other ones.

References

- [1] N. Aggarwal, P. Ranganathan, N.P. Jouppi, and J.E. Smith. Isolation in Commodity Multicore Processors. *Computer*, 40(6):49–59, June 2007.
- [2] Gene M. Amdahl. *Validity of the single processor approach to achieving large scale computing capabilities*, pages 79–81. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2000.
- [3] J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent Programming in Erlang*. Prentice Hall, 2 edition, 1996.
- [4] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report UCB/EECS-2006-183, Electrical Engineering and Computer Sciences, University of California at Berkeley, December 2006.

- [5] G.H. Barnes, R.M. Brown, M. Kato, D.J. Kuck, D.L. Slotnick, and R.A. Stokes. The ILLIAC IV Computer. *IEEE Transactions on Computers*, 17(8):746–757, 1968.
- [6] Bill Blume, Rudolf Eigenmann, Keith Faigin, John Grout, Jay Hoe, David Padua, Paul Petersen, Bill Pottenger, Lawrence Rauchwerger, Peng Tu, and Stephen Weatherford. Polaris: The next generation in parallelizing compilers. In *Proceedings of the Workshop on Languages and Compilers for Parallel Computing*, pages 10–1. Springer-Verlag, Berlin/Heidelberg, 1994.
- [7] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA, 1996.
- [8] F. Boussinot and R. de Simone. The ESTEREL language. *Proceedings of the IEEE*, 79(9):1293–1304, Sep 1991.
- [9] Doug Burger, James R. Goodman, and Alain Kägi. Memory bandwidth limitations of future microprocessors. pages 78–89, 1996.
- [10] J. Choi and H. Srinivasan. Deterministic Replay of Java Multithreaded Applications. In *Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools*, pages 48–59, August 1998.
- [11] David E. Culler and Richard M. Karp and David A. Patterson and Abhijit Sahay and Klaus E. Schauser and Eunice Santos and Ramesh Subramonian and Thorsten von Eicken. LogP: Towards a Realistic Model of Parallel Computation. In *Principles Practice of Parallel Programming*, pages 1–12, 1993.
- [12] Susan J. Eggers, Joel S. Emer, Henry M. Levy, Jack L. Lo, Rebecca L. Stamm, and Dean M. Tullsen. Simultaneous multithreading: A platform for next-generation processors. *IEEE Micro*, 17(5):12–19, 1997.
- [13] Joel Emer, Mark D. Hill, Yale N. Patt, Joshua J. Yi, Derek Chiou, and Resit Sendag. Single-threaded vs. multithreaded: Where should we focus? *IEEE Micro*, 27(6):14–24, Nov/Dec 2007.
- [14] M.J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, 21:948–960, 1972.
- [15] Steven Fortune and James Wyllie. Parallelism in random access machines. In *STOC '78: Proceedings of the tenth annual ACM symposium on Theory of computing*, pages 114–118, New York, NY, USA, 1978. ACM.
- [16] High Performance Fortran Forum. High Performance Fortran Specification Version 2.0, January 1997.
- [17] MPI Forum. MPI-2: Extensions to the Message-Passing Interface. Technical report, University of Tennessee, Knoxville, Tennessee, July 1997.
- [18] Ian Foster. *Designing and Building Parallel Programs*. Addison-Wesley, 1995.
- [19] Keir Fraser and Tim Harris. Concurrent programming without locks. *ACM Trans. Comput. Syst.*, 25(2):5, 2007.

- [20] N. H. Gehani and W. D. Roome. Concurrent C. *Software-Practice and Experience*, 16:821–844, 1986.
- [21] Richard Goering, John Gustafson, Arvind, Gary Smith, Kunle Olukotun, Gene Amdahl, and Patrick H. Madden. "Can We Still Keep the Faith?": A debate on the Future of Multi-Core Systems. In *ACM/SIGDA Dinner and Open Member Meeting*, page 3, 2007.
- [22] John L. Gustafson. Reevaluating Amdahl's law. *Commun. ACM*, 31(5):532–533, 1988.
- [23] Michael R. Head and Madhusudhan Govindaraju. Approaching a Parallelized XML Parser Optimized for Multi-Core Processors. In *Workshop on Service-Oriented Computing Performance: Aspects, Issues, and Approaches , held in conjunction with IEEE International Symposium on High Performance Distributed Computing (HPDC) 2007*, Monterey Bay, California, June 2007.
- [24] Mark D. Hill. What is scalability? *SIGARCH Comput. Archit. News*, 18(4):18–21, 1990.
- [25] Mark D. Hill and Michael R. Marty. Amdahl's law in the multicore era. *Computer*, 41(7):33–38, 2008.
- [26] Ravi Iyer, Li Zhao, Fei Guo, Ramesh Illikkal, Srihari Makineni, Don Newell, Yan Solihin, Lisa Hsu, and Steve Reinhardt. QoS policies and architecture for cache/memory in CMP platforms. *SIGMETRICS Perform. Eval. Rev.*, 35(1):25–36, 2007.
- [27] Anita K. Jones and Peter Schwarz. Experience using multiprocessor systems—a status report. *ACM Comput. Surv.*, 12(2):121–165, 1980.
- [28] Geraint Jones. *Programming in Occam*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1988.
- [29] Ralf Lämmel. Google's MapReduce programming model – Revisited. *Sci. Comput. Program.*, 70(1):1–30, 2008.
- [30] James R. Larus and Ravi Rajwar. *Transactional Memory*. Morgan & Claypool Publishers, San Rafael, CA, USA, 2006.
- [31] Wei Lu and Dennis Gannon. ParaXML: A Parallel XML Processing Model on the Multicore CPUs. In *Technical Report TR662. School of Informatics*, Indiana University, April 2008.
- [32] Deborah T. Marr, Frank Binns, David L. Hill, Glenn Hinton, , David A. Koufaty, J. Alan Miller, and Michael Upton. Hyper-Threading Technology Architecture and Microarchitecture. *Intel Technology Journal*, 6(1), 2002.
- [33] Timothy G. Mattson, Beverly A. Sanders, and Berna L. Massingill. *Patterns for Parallel Programming (Software Patterns Series)*. Addison-Wesley Professional, 1st edition, September 2004.
- [34] D. May, R. Shepherd, and C. Keane. Communicating Process Architecture: Transputers and Occam. In D. B. Skillicorn and D. Talia, editor, *Programming Languages for Parallel Processing*, pages 92–105. IEEE Press, 1995.
- [35] Richard McDougall. Extreme software scaling. *ACM Queue*, 3(7):36–46, 2005.
- [36] Bill Ogden, Luiz Fadel, and Roger Fowler. IBM zSeries 990 Technical Introduction, May 2003.

- [37] Kunle Olukotun and Lance Hammond. The future of microprocessors. *ACM Queue*, 3(7):26–29, 2005.
- [38] Gregory F. Pfister. *In Search For Clusters*. Prentice Hall Ptr, 2 edition, 1997.
- [39] N. Rafique, Won-Taek Lim, and M. Thottethodi. Effective Management of DRAM Bandwidth in Multicore Processors. *Parallel Architecture and Compilation Techniques, 2007. PACT 2007. 16th International Conference on*, pages 245–258, Sept. 2007.
- [40] Rohit Chandra and Ramesh Menon and Leo Dagum and David Kohr and Dror Maydan and Jeff McDonald. *Parallel Programming in OpenMP*. Morgan Kaufmann, October 2000.
- [41] Wolfgang Schreiner. *Parallel Functional Programming An Annotated Bibliography*, August 1997.
- [42] L. Spracklen and S. G. Abraham. Chip multithreading: Opportunities and challenges. In *11th International Symposium on High-Performance Computer Architecture (HPCA-11)*, pages 248–252, 2005.
- [43] John A. Stratton, Sam S. Stone, and Wen mei W. Hwu. MCUDA: An Efficient Implementation of CUDA Kernels for Multi-Core CPUs. *Proceedings of the 21st International Workshop on Languages and Compilers for Parallel Computing*, August 2008.
- [44] Herb Sutter. A Fundamental Turn Toward Concurrency in Software. *Dr. Dobbs's Journal*, mar 2005.
- [45] Herb Sutter and James Larus. Software and the concurrency revolution. *ACM Queue*, 3(7):54–62, 2005.
- [46] Andrew S. Tanenbaum. *Distributed Operating Systems*. Prentice Hall, 1995.
- [47] Christian Tismer. Continuations and Stackless Python, or How to change a paradigm of an existing program. In *In Proceedings of the 8th International Python Conference*, 1999.
- [48] Jay Trodden and Don Anderson. *HyperTransport(TM) System Architecture (PC System Architecture Series)*. Addison-Wesley Professional, February 2003.
- [49] Peter Tröger, Martin von Löwis, and Andreas Polze. The Grid-Occam Project. In *GSEM 2004 - Grid Services Engineering and Management, First International Conference*, pages 151–164, Erfurt, Germany, September 2004. Springer, Lecture Notes in Computer Science.
- [50] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *ISCA '95: Proceedings of the 22nd annual international symposium on Computer architecture*, pages 392–403, New York, NY, USA, 1995. ACM.
- [51] Theo Ungerer, Borut Robič, and Jurij Šilc. A survey of processors with explicit multithreading. *ACM Comput. Surv.*, 35(1):29–63, 2003.
- [52] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, 1990.
- [53] Duc Vianney. Hyper-Threading speeds Linux. *IBM DeveloperWorks*, January 2003.